NPS52-83-005

NAVAL POSTGRADUATE SCHOOL

Monterey, California

ON DEADLOCK DETECTION IN DISTRIBUTED COMPUTING SYSTEMS

D. Z. Badal and M. T. Gehl

April 1983

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

David A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-83-005	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) On Deadlock Detection in Distributed Computing Systems		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dushan Z. Badal and LCDR Michael T. Gehl, USN		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152M;RR000-01-10 N0001483WR30104
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE April 1983 13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Chief of Naval Research Arlington, Va 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This paper has been published in the Proceedings of the 2nd Joint Conference of the IEEE Computer and Communication Societies, INFOCOM83, San Diego, CA, April 1983.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Deadlock Detection, Distributed Computing Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

With the advent of distributed computing systems, the problem of deadlock, which has been essentially solved for centralized computing systems, has reappeared. Existing centralized deadlock detection techniques are either too expensive or they do not work correctly in distributed computing systems. Although several algorithms have been developed specifically for distributed systems, the majority of them have also been shown to be inefficient or incorrect. A new algorithm is proposed which is more efficient than any existing distributed deadlock detection algorithm.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ON DEADLOCK DETECTION IN DISTRIBUTED COMPUTING SYSTEMS

D. Z. Badal and M. T. Gehl

Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

ABSTRACT

With the advent of distributed computing systems, the problem of deadlock, which has been essentially solved for centralized computing systems, has reappeared. Existing centralized deadlock detection techniques are either too expensive or they do not work correctly in distributed computing systems. Although several algorithms have been developed specifically for distributed systems, the majority of them have also been shown to be inefficient or incorrect. A new algorithm is proposed which is more efficient than any existing distributed deadlock detection algorithm.

I. INTRODUCTION

Deadlock is a circular wait condition which can occur in any multiprogramming, multiprocessing or distributed computer system which uses locking if resources are requested when needed and processes are not assigned priorities. It indicates a state in which each member of a set of transactions is waiting for some other member of the set to give up a lock. An example of a simple deadlock is shown in Figure 1. Transaction T1 holds a lock on resource R1 and requires resource R2; transaction T2 holds a lock on resource R2 and requires R1. Nei-

ther transaction can proceed, and neither will release a lock unless forced by some outside agent. There have been many algorithms published for deadlock detection, prevention or avoidance in centralized multiprogramming systems. The problem of deadlock in those systems has been essentially solved. With the advent of distributed computing systems, however, the problem of deadlock reappeared. Certain peculiarities of distributed systems (lack of global memory and non-neglibible message delays, in particular) make centralized techniques for deadlock detection expensive. Recently there have been published several deadlock detection algorithms for distributed systems [OBE82, GLI80, MEN79, GRA78, TSA82]. However, most of them have been shown to be incorrect or to be too complex and expensive to be practical. In this paper, we propose a new distributed deadlock detection algorithm for distributed computing systems which is more efficient than any other published deadlock detection algorithm. The major differences between the proposed algorithm and existing algorithms are the concept of a Lock History which each transaction carries with it, the notion of Intention Locks and a three staged approach to deadlock detection, with each stage, or level, of detection activity being more complex than the preceding. In this paper, we first present the algorithm, then an informal proof of correctness, and finally a performance comparison of the proposed algorithm with the algorithm presented in [OBE82]. Obermarck's algorithm is used for comparison for two reasons. First, it is the most recently published distributed deadlock detection algorithm and it is also shown to be more efficient than other algorithms.

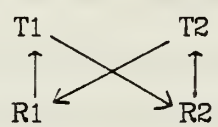

Fig. 1 -- A simple deadlock cycle

Second, Obermarck's algorithm is being implemented on IBM's distributed database System R*.

II THE PROPOSED ALGORITHM

A. INTRODUCTION

The proposed algorithm assumes two types of locks: Exclusive Write(W) and Shared Read(R). Additionally, the proposed algorithm uses an Intention Lock (I) which indicates that a transaction wishes to acquire a lock on a resource, either to modify it (IW) or to read it (IR). The Intention Locks are placed in a resource Lock Table when an agent is created at a site of a locked resource which it requires, or when a resource at the same site is requested but is already locked by another transaction. The Intention Locks are also placed in the Lock Table of the last locked resource(s) once the transaction can determine which resource(s) intends to lock in its next execution step. The Intention Locks are not the same as the Intention Modes used by Gray when he discusses hierarchical locks in [GRA78]. Gray uses the Intention Mode to "tag" ancestors of a resource in a hierarchical set of resources as a means of indicating that locking is being done on a "finer" level of granularity, and therefore preventing locking on the ancestors of the resource. The rules for locks in the proposed algorithm are the same as for conventional locking, i.e., any number of transactions or agents may simultaneously hold Shared Read Locks on a particular resource, but only a single transaction or agent may hold an Exclusive Write Lock on a resource. Any number of Intention Locks (IW or IR) may be placed on a resource, which means that any number of transactions may wait for a resource. Each site must therefore have some method for determining which transaction will be given the resource when it becomes free. Our algorithm uses Lock History (LH) of a transaction which is a record of all types of locks on any resources

which have been requested or are held by that transaction. An example of a Lock History for transaction T1 is LH(T1): {W(R3C), W(R2B), R(R1A)}. This LH shows that T1 holds a Write Lock on resource R3 at site C, a Write Lock on resource R2 at site B, and a Read Lock on resource R1 at site A. The information contained in a Lock Table for a resource includes a) the transaction or agent ID and its Lock History, b) the type of lock and c) the resource (and type of lock) which that transaction holding this lock intends to lock next. The field containing the current lock will be referred to as the "current" field of the Lock Table, and the field containing the future intentions of that transaction holding the "current" lock will be called the "Next" field. For clarity, Lock Histories will be shown as separate entities. An example of a Lock Table is LT(R2B): T1{W(R2B), IW(R3C)}; T2{IW(R2B)}. The Lock Table for resource R2 at site B shows that T1 holds a Write Lock on R2, and that T2 has placed an Intention Write Lock on R2. T1 has also indicated that it intends to place a Write Lock on resource R3 at site C. The proposed algorithm assumes a distributed model of transaction execution where each transaction has a Site of Origin (Sorig), which is the site at which it entered the system. Whenever a transaction requires a remote resource, (a resource at a site other than the site it is currently at), it "migrates" to the site where that resource is located. Migration consists of creating an "agent" at the new site. The transaction agent then executes, and may either create additional agents, start commit or abort actions, or return execution to the site from which it migrated. This transaction model is consistent with recent literature [OBE82, GRA81B]. When a transaction migrates, it carries along its Lock History. A Wait-For Graph (WFG) is constructed by the deadlock detection algorithm, using the Lock Histories of transactions which are possibly involved in a deadlock cycle, any time a transaction or agent attempts to place a lock on a resource which is already locked, or when it determines that a remote resource will be required. There are two types of nodes in the WFG;

transactions (or agents) and resources. A directed arc from a resource node to a transaction node indicates that the transaction has a lock on the resource, while a directed arc from a transaction node to a resource indicates that the transaction has placed an Intention Lock on that resource. A cycle in the TWG indicates the existence of the deadlock. The WFS is a list of transaction - waits - for - transaction strings (obtained from the site's WFG), in which each transaction is waiting for the next transaction in the string, and the Lock History for each transaction in the string. For example, the WFS [T1{W(R2A), IW(R3B)}, T4{W(R3B)}] shows that T1 is waiting for T4, and each transaction's Lock History is in brackets. A transaction may also bring along other information such as a metric representing its execution cost, but such information is not included in this paper as it is outside the primary function of the proposed deadlock detector. We assume that each transaction or agent will have a globally unique identifier which indicates its Site of Origin. Agents can be in any of three states; active, blocked (waiting), or inactive. An inactive agent is one which has done work at a site and created an agent at another site or returned execution to its creating site, and is now awaiting further instructions, such as commit, abort or become active again. A blocked transaction is one which has requested a resource which is locked by another transaction. An active agent is one which is not blocked or inactive. To allow concurrent execution, a transaction may have several active agents. Each site in the system has a distributed deadlock detector, which performs deadlock detection for transactions or agents at that site. Several sites can simultaneously be working on detection of any potential deadlock cycle. The basic premise of the proposed algorithm is to detect deadlock cycles with the least possible delay and number of inter-site messages. Based on the findings by Gray and others [GRA81A] that cycles of length 2 occur much more frequently that cycles of length 3, and cycles of length 3 occur much more frequently that cycles of length 4, and so on, the proposed algorithm uses

a staged approach to deadlock detection. We distinguish two types of deadlock cycles to be considered; a) those which can be detected using only the information available at a site, and b) those which require inter-site messages to detect. In the proposed algorithm, the first type has been divided into two levels of detection activity. Because the proposed algorithm checks for possible deadlock cycles every time a remote resource is requested and another transaction is waiting for a resource being released by the transaction making the remote resource request or a local resource is requested but already locked, the level one check should be as quick as possible. If the requested resource is still not available "after X units of time" [GRA78], then the probability of a deadlock has increased sufficiently to justify a more complex and time-consuming check in level two. Therefore the proposed algorithm has three levels of deadlock detection activity. Levels one and two correspond to the first type of deadlock cycle, while level three corresponds to the second type. The first level is designed to detect cycles of length 2, although certain more complex deadlock cycles could be detected, depending on the topology of the deadlock cycle. This level uses only information available in the Lock Table of the requested resource if the resource is local, or the last locked resource if the requested resource is at another site, and in the transaction Lock Histories. Due to the information contained in the "Next" field of the Lock Table and in each transaction's Lock History, this level of detection activity can detect all direct deadlocks of cycle length 2 involving one or two sites. The deadlock is direct if at least one transaction is blocked, i.e., is waiting for the resource R locked by another transaction T' and R is the last resource locked by T' before it becomes blocked too. The direct deadlocks occur mostly due to locks being released after the transaction does not require the resource any more. The indirect deadlocks can occur when the resources are kept locked until the transaction termination as is done in database systems which use two-phase locking.

As an example, let transaction T1 at site A Write Lock resource R1. Let transaction T2 at site B Write Lock resource R2. These locks would be placed in the Lock Tables of the respective resources, and also in the Lock Histories for the respective transactions. Transaction T1 now determines that it must lock a remote resource R2, so it places that information in the "Next" field of its lock entry of resource R1 and in its Lock History. It then migrates to site B, where its agent places an Intention Lock in the Lock Table for R2, and then becomes blocked, waiting for resource R2 to be released. A level one check is made using the Lock Table of R2, showing no deadlock cycles. Now transaction T2 determines that it requires a Write Lock on a remote resource R1. It places that information in the "Next" field of its lock entry in the Lock Table of R2 and in its Lock History. As T1 is waiting for R2 a deadlock detector triggers level one of the deadlock detection algorithm before T2 migrates to site A. The deadlock detection algorithm combines the Lock Histories of all transactions holding or requesting locks on R2 (T1 and T2) into a WFG, and detects a deadlock. In this example, the cost of creating an agent of T2 at site A was saved by a very quick check for cycles of length two. Inasmuch as the majority of deadlocks occurring will be of this length, this simple and inexpensive check will detect the majority of deadlocks as they occur. If, in the example just given, transactions T1 and T2 had simultaneously determined the need for locks at the other site, the initial level one check would not have been performed because no transactions were waiting for those resources. Both transactions would have migrated and placed Intention Locks at the new sites. A level one check is then made at each site when it is noted that the requested resource is not available. Each site constructs a WFG from the Lock Histories of the transactions in the Lock Tables of the requested resources, and each site will detect a deadlock cycle in the WFG without any inter-site messages. Even if the first level of detection activity fails to detect a deadlock cycle, there can still be a more complex deadlock cycle in existence.

The second level of detection activity requires more time because it constructs a WFG using all Lock information available at the site, i.e., Lock information from all resource Lock Tables at the site. If we assume that more complex deadlock cycles are comparatively rare, it is advantageous to "wait X units of time" [GRA78] before starting the second level of detection activity. If a transaction is still waiting to acquire a lock after these X units of time, the probability of a more complex deadlock cycle existing has increased sufficiently to justify a more comprehensive check. As previously mentioned, the second level still attempts to detect a cycle using information available at the same site where the transaction is waiting for a resource. The Lock Histories of all blocked or inactive transactions at the site, and the Lock Histories from all transactions in the WFSs from other sites are combined into a new Wait-For Graph. (The WFS's are generated by the third level of the proposed algorithm). If no deadlock is detected, and because level three of deadlock detection activity involves inter-site communication, it might be advantageous to wait Y units of time before continuing in order to increase the probability of the wait condition being an actual deadlock. After Y units of time, when the deadlock detection algorithm is ready to continue, the WFG is converted into a WFS. The WFS is then sent to other sites. The version of the algorithm presented here includes an optimization whereby the WFS is sent to the site to which the transaction being waited for has migrated only if the first transaction in the WFS has a higher lexical ordering than the transaction which has migrated. This optimization is similar to one used in [OBE82]. When a site deadlock detector receives a WFS, it substitutes the latest Lock Histories for any transaction for which it has a later version (the longest Lock History is the latest). It then constructs a new WFG and checks for cycles. If a cycle is found, it must be resolved. If any transactions are waiting for other transactions which have migrated to other sites, the current site must repeat the process of constructing WFG's and sending them to the sites to which

the transactions being waited for have migrated, subject to the constraints of the optimization. If the transactions being waited for are at this site and active, deadlock detection activity can cease. Level three activity will continue until a deadlock is found or it is discovered that there is no deadlock. The following definitions are used in the description of the algorithm:

IL – Intention Lock
W(v) -- Exclusive Write lock on resource x
R(x) – Shared Read lock on resource x
IW(x) – Intention Lock(Write) on resource x
IR(x) – Intention Lock(Read) on resource x
Sorig(T) – Site or Origin of transaction T
LT(R) -- Lock Table for resource R
LH(T) – Lock History for transaction T
"Next" – Field in Lock Table reflecting the resource the transaction intends to acquire next
"Current" – Field in Lock Table reflecting the lock currently held by a transaction

B. THE ALGORITHM

1. {Remote resource R requested or anticipated by transaction or agent T}

   A. Place appropriate IL entry in "next" field of the Lock Table of the current resource (the last resource locked by T,if any) and in LH(T).

   B. {Start level 1 detection activity at current site}. If another transaction is waiting for the last resource locked by T, construct a Wait-For graph and WFS from the Lock Histories of the transactions holding and requesting that resource and check for cycles.

   C. If no cycles are detected or if no transactions are waiting:

      1) Collect LH(T) and the WFS (generated at step 1.B) from the current site, and have an agent created at the site of the requested resource.

      2) Stop

   D. If a cycle is detected, resolve the deadlock

2. {Local resource R requested}

   A. If resource R is available: {Lock it}

      1) Place appropriate lock in Lock Table of resource R and in LH(T).

2) end

B. If resource is not available: {Start level 1 detection activity}

1) Place appropriate IL in Lock Table of resource R and in LH(T).

2) Construct a WF Graph from Lock Histories of all transactions holding and requesting R, and check for cycles.

3) If there are no cycles, and if the transaction holding the lock on R is still at this site and active, stop. If there is a cycle, resolve the deadlock.

4) If the transaction holding the lock on R has either migrated to another site, or is still at this site but is blocked by another transaction which has migrated to another site, delay(t1).

5) If resource is now available:

a) Remove IL from Lock Table and LH(T)

b) Go to step 2A

6) If resource is not available: {Start level 2 activity}

a) Construct a WFG using the Lock Histories of the transactions in the WFSs which have been sent from other sites by level three detection activity, and the Lock Histories of all blocked or inactive transactions at this site and check for cycles.

b) If any cycles are found, resolve the deadlock.

c) If no cycles are found, Delay(t2)

d) If the requested resource is now available, go to step 2A

e) If the transaction being waited for is at this site and active, stop.

f) If the resource is still not available, go to step 3 {Start level 3 detection activity}.

3. {Wait-For Message Generation}

A. {Start Level 3 detection activity} Construct a WFS by condensing the latest WFG into a list of strings of transactions waiting for transactions. Add the Lock Histories of each transaction in string.

B. Send the WFS to the site to which the transaction being waited for has gone if the transaction being waited for in each substring has a smaller identifier than the first transaction in that substring.

4. {Wait-For Message Received}

A. {Start level 3 detection activity} Construct a WFG from the Lock Histories of the transactions in the WFS's from other sites, and from the Lock Histories of all blocked or inactive transactions at this site. (Use the latest WFS from each site.)

B. If this WFG shows that a transaction which is being waited for has migrated to another site, go to step 3. {Repeat WFS Generation}

C. If the transaction being waited for is active, and has not indicated by an Intention Lock that it will attempt to acquire a resource which may result in a deadlock, discard the WFG and stop.

D. If the transaction being waited for is active but has indicated by an Intention Lock that it is going to a site which will cause a deadlock, or if a cycle is found, resolve the deadlock.

•

C. EXPLANATION OF THE ALGORITHM

Step 1. This step is executed any time a transaction (or agent) T requests a remote resource, or when it determines that it will require a remote resource. The Lock Table of the resource which the transaction is currently using (or has just finished with) is checked to see if any other transactions are waiting (i.e., have placed Intention Locks) for that resource. If so, the Lock Histories of all transactions requesting and holding the resource are combined into a WFG and a check for cycles is made. If no cycle is found, T collects the WFS formed from the WFG at that site and causes an agent to be created at the site of the requested resource. Step 2. This step is executed each time a local resource is requested, either by an agent (transaction) already at that site or by a newly created agent. If the resource is available, appropriate locks are placed and the resource granted. If the resource is not available, Intention Locks are placed in the Lock Table of the requested resource and in the Lock History of the

requesting transaction, a WFG is constructed using only the information in the Lock Table of the requested resource and the Lock Histories of the transactions holding or requesting that resource, and a quick level one check is made for possible deadlock cycles. If no cycles are found, the algorithm waits for a certain period of time before continuing. This should allow the transaction which holds the resource to complete its work and release the resource. If the resource is not available after this delay, the chance of a deadlock is higher, so the algorithm shifts to another level of detection. It now uses the Lock Histories from each blocked or inactive transaction at the site, as well as from any WFS's from other sites which have been brought by migrating transactions. If there are no cycles in this graph, and the resource is still not available after a second delay (also tunable by the system users), the possibility of deadlock is again much greater, but the current site has insufficient information to detect it. Therefore the proposed algorithm progresses to the third level of detection (step 3). Step 3. The Wait-For message generated by this step consists of a collection of substrings. Each substring is a list of transactions each of which is waiting for the next transaction in the substring. The substring also contains the resources Locked or Intention Locked by each transaction in the substring. This step includes the optimization that a WFS is only sent to another site if the transaction which has migrated has a lower lexical ordering than the first transaction in the substring. For example, for the WFG shown in Figure 2, the WFS would be [T2{W(R2B),IW(R3C)}, T3{W(R3C), IW(R4D)}, T4{W(R4D),IW(R1A)}]. T4 has migrated to site A. The WFS would be sent to site A only if T4 is less than T2.

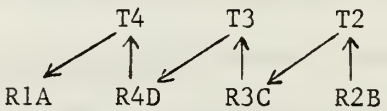

Fig. 2 -- Example WFG

Step 4. In this step, the Lock Histories of the transactions in the WFS's previ-

ously received from other sites, and the Lock Histories of any blocked or inactive transactions at this site are added to the Wait-For information contained in a received WFS. If there is still insufficient information to detect a cycle (a transaction being waited for has migrated to another site), another iteration must be performed, so the algorithm repeats by transferring to step 3. If a cycle is detected, it is resolved, and if the last transaction being waited for is still active, the algorithm stops.

D. OPERATION OF THE ALGORITHM

The operation of the algorithm will be shown by executing it on the following example. T1 migrates to site B and locks resource R2. It then migrates to site C and locks resource R3. T4 locks resource R4 at site D. At this point, the Lock Histories and Lock Tables are as in Fig. 3.

T1 now attempts to acquire resource R4. By step 1, an IL entry is placed in

Site A

LH(T1): {IW(R2B)}

Site B

LH(T1): {W(R2B),IW(R3C)}

LT(R2B): T1{W(R2B)}

Site C

LH(T1): {W(R2B),W(R3C)}

LT(R3C): T1{W(R3C)}

Site D

LH(T4): {W(R4D)}

LT(R4D): T4{W(R4D)}

Fig. 3 Lock Histories and Lock Tables

LH(T1) and in LT(R3) at site C. As there are no Intention Locks in LT(R3C), the WFS from site C is collected (at this point in time, none exists), and an agent of T1 is created at site D, with T1 "bringing" LH(T1): {W(R2B), W(R3C), IW(R4D)}. Site D now applies step 2B1, and places the IL entry in LT(R4D) and LH(T1). Then it executes step 2B2 by combining the Lock Histories of T1 and T4. No cycles are found, but as T4 is still active at site D, the DDA is stopped. The current status of the Lock Tables and Lock Histories is as in Fig. 4. T4 now determines that it needs to write into resource R3. It applies step 1 and places an IL entry in LH(T4) and LT(R4D). The Lock Table for R4 is now LT(R4D): T4{W(R4D),IW(R3C)}; T1{IW(R4D)}, and the Lock History for T4 is now LH(T4): {W(R4D), IW(R3C)}. It sees in LT(R4D) that T1 is waiting for R4, so it combines its Lock History with T1's. This reflects the cycle T1-->T4-->T1, so a deadlock has been detected with no intersite messages.

Site A

LH(T1): {IW(R2B)}

Site B

LH(T1): {W(R2B),IW(R3C)}

LT(R2B): T1{W(R2B)}

Site C

LH(T1): {W(R2B),W(R2C),IW(R4D)}

LT(R3C): T1{W(R3C),IW(R4D)}

Site D

LH(T4): {W(R4D)}
LH(T1): {W(R2B),W(R3C),IW(R4D)}

LT(R4D): T4{W(R4D);T1{IW(R4D)}

Fig. 4 Lock Tables and Lock Histories

III. INFORMAL PROOF OF CORRECTNESS

In general, a deadlock cycle can have many different topologies. For the model of transaction execution used in the proposed algorithm (migration of agents of transactions), these different topologies can be loosely grouped into four categories. Category A involves local deadlocks in which all the resources and transactions involved in the deadlock are local, i.e., located at one site, and thus the transactions involved have not locked any resources at other sites. Category B is the same as category A, with the exception that the transactions are nonlocal, i.e., they may have locked resources at other sites. Category C contains direct deadlocks of cycle length two involving only one transaction and one resource at each of two sites. Category D is a generalization of category C deadlocks; any number of transactions and resources may be directly or indirectly deadlocked at any number of sites. For each category, it will be argued that the algorithm detects all possible deadlocks in that category, and that the algorithm does not detect "false" deadlocks except in the case where a transaction which was involved in a deadlock has aborted, but its agents have not yet been notified. If all the transactions and resources involved in a deadlock are located at the same site and none of the transactions have locked resources at other sites, each transaction's Lock History will be an accurate and complete snapshot of the locks placed by that transaction. If the deadlock cycle length is two, the combination of the Lock Histories in step 2B2 (level 1) will detect the cycle. If the length of the cycle is greater than two, step 2B6 (level 2) will combine, for this category of deadlock cycles, the Lock Histories of all the blocked or inactive transactions at the site. This information will be a complete and accurate global snapshot of the deadlock cycle, and hence the deadlock will be detected. Deadlocks in the second category are those in which all the transactions and resources involved are at one site, but the transactions involv

may have locked resources at other sites before creating the agent at this site. The argument to show that all deadlocks in this category will be detected by the proposed algorithm is essentially the same as the one used for the first category. Since all the transactions involved in the deadlock are currently at this site, their Lock Histories are complete and accurate in so far as they pertain to the deadlock cycle. It is possible, in the case of concurrent execution of a transaction's agents, for an agent involved in a deadlock to be unaware of resources locked by other agents of that transaction which are executing concurrently, and will probably still be active. The only difference between this case and the preceding is that the WFGs constructed by steps 2B2 and 2B6 may contain information about other locks held by the transactions involved, but the information concerning the deadlock cycle will be present. Deadlocks in the third category will be detected by level 1 because a single Lock Table at each site holds sufficient information to detect a deadlock cycle. If the migrations occur simultaneously, the "Next" field of the Lock Table of the requested resource would show an Intention Lock on the other resource, and this cycle would be detected by step 2B2. If the migrations occurred sequentially, the second transaction would, before migrating, place an Intention Lock in the Lock Table of its last locked resource. The level 1 check of step 1B would cause a WFG to be constructed which would reveal the deadlock cycle. The fourth category of deadlock cycles is a generalization of the third. Deadlock cycles in this category may involve any number of transactions and resources at any number of sites. A record is always kept of the site to which a transaction has migrated (in the "Next" field of it's last locked resource at the current site.) If level 2 cannot detect the cycle in step 2B6 with information at that site, level 3 causes a WFS containing this site's information to be sent to the site to which the transaction has migrated if the transaction which has migrated has a lower unique identifier than the first transaction in the substring. Steps 3 and 4 cause this process to

be continued, with each site adding additional information, until a site contains enough information to detect a deadlock cycle or determine that no deadlock exists, regardless of the number of migrations made by a transaction. To show that this process will continue until the deadlock is detected, we refer to the proof in [OBE82], since the optimization in the proposed and in the Obermarck's algorithm is essentially the same. False deadlocks are an anomaly where a non-existent deadlock cycle is detected by a deadlock detection algorithm, and are usually a result of incorrect or obsolete information. Since the proposed algorithm uses only the latest copy of a transaction's Lock History for deadlock detection purposes, the information used cannot be incorrect in the sense of invalid entries, although it may be incomplete. This means that a Wait-For graph constructed from incomplete versions of Lock Histories may have insufficient information to detect a deadlock at that particular level of detection activity or iteration of level three activity, but it will not have incorrect information. When a transaction which has agents at two or more sites commits or aborts, however, it is possible that the commit or abort messages to other agents of that transaction may be delayed. Obviously, a transaction which is ready to commit cannot have any of it's agents in a blocked state (and therefore in a possible deadlock condition), so its agents can either be only active or inactive. While inactive agents may be being waited for by agents of other transactions, no Lock History or Lock Table can show that an agent of the transaction which is about to commit is waiting for another transaction, so no false deadlocks can exist. Therefore only the possibility of a transaction which is in the process of aborting and thus causing a false deadlock to be detected must be considered. Suppose an agent of a transaction decides to abort, but before its abort message reaches another agent of that transaction, a deadlock is found involving that transaction. Technically, this could be considered a false deadlock, since one of the transactions involved has aborted, probably breaking the deadlock cycle. If the

deadlock cycle is complex, and the proposed algorithm is performing level two or three detection activity, the delays introduced in steps 2B4 and 2B6c should allow the abort message to arrive. For what we believe to be the rare occurences where the abort message does not arrive, it would probably be more efficient to let the deadlock detection algorithm resolve the (false) deadlock rather than having the algorithm perform some explicit action (such as delaying before resolving any detected deadlock cycle) each time it detects a deadlock.

IV. PERFORMANCE ANALYSIS

To check the efficiency (in terms of inter-site messages) of the algorithm, it was analyzed in several deadlock scenarios. The algorithm of Obermarck [OBE82] was also analyzed in these scenarios. Obermarck's algorithm was chosen for this comparison because it is being implemented in IBM's developmental distributed database system, System R* and because its performance has been shown [OBE82] superior to other deadlock detection algorithms. Since the majority of deadlocks which will occur will be of length two or three, three test cases involving deadlock cycles of those lengths will be used for the comparison. It is assumed that the transactions are lexically ordered T1 < T2 < T3. These cases are shown in Figure 5. T1 originated at site A and holds a lock on R1, and T2 originated at site B and holds a lock on R2. In cases two and three, T3 originated at site C and holds a lock on R3. In case one, T1 has migrated to site B and requested R2, while T2 has migrated to site A and requested R1. In case two, T1 has migrated to site B and requested R2, T2 has migrated to site C and requested R3, and T3 has migrated to site A and requested R1. In case three, T1 has migrated to site C and requested R3, T2 has migrated to site A and requested R1, and T3 has migrated to site B and requested R2.

For case one, where the deadlock cycle is of length two, the proposed algorithm

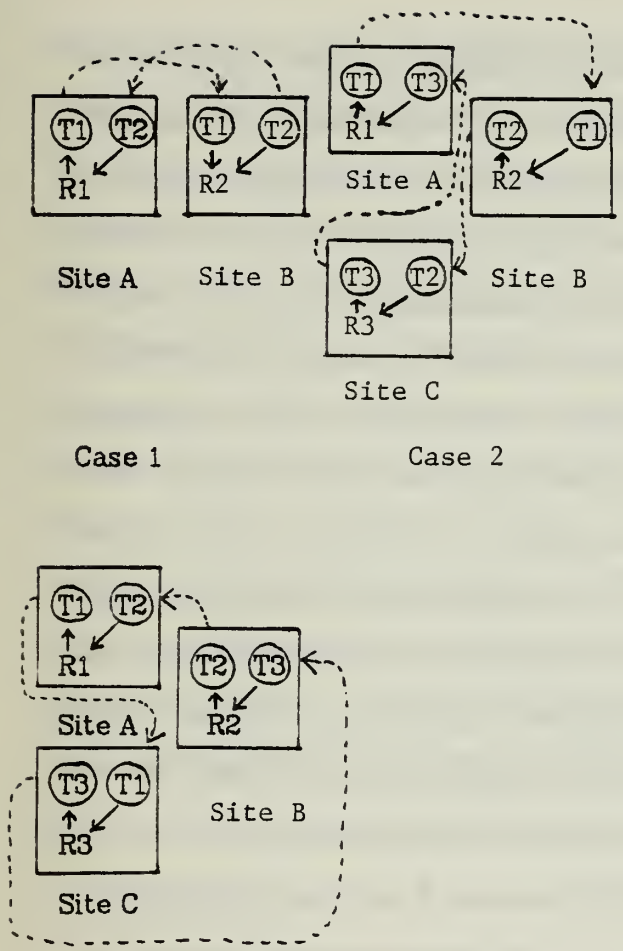

Fig. 5 – Deadlock cycles used in performance analysis

requires no additional messages for deadlock detection, while Obermarck's algorithm requires one message. For case two, with a deadlock cycle of length three, Obermarck's algorithm requires two messages. The number of messages required by the proposed algorithm is dependent on the timing of the transaction migrations. If the migrations occur at different times (i.e., sequentially), no messages are required. If, however, the migrations happen to occur simultaneously, only one message is generated because of the optimization. A similar situation occurs in case three. If the migrations occur simultaneously, two messages will be generated by the proposed algorithm, although one of these mes-

sages is redundant, i.e., any one message is sufficient to detect deadlock. If transaction migrations occur at different times (i.e., sequentially) then no messages are required. Obermarck's algorithm requires three messages, regardless of the timing of the migrations. As pointed out in [OBE82] it is apparent that in the overwhelming majority of cases the global deadlocks are of cycles of length two involving two sites. No messages are required to detect direct global deadlocks of cycle length two by the proposed algorithm. In order to provide the evaluation of both algorithms for global deadlocks with cycle length $n > 2$ we assume that n nonlocal (or global) transactions are involved in the global deadlock such that at each of n sites only one transaction is blocked by another transaction and each transaction needs to execute only at two sites. Then the number of messages needed by the proposed algorithm for the worst case scenario (when all the transactions involved migrate simultaneously) can be shown to be N-1, where $N = \sum_{k=1}^{n}(n-k)$. Under the same circumstances it can be shown that Obermarck's algorithm [OBE82] requires N messages regardless of sequencing of transaction migrations, i.e., the number of messages depends only on the number of transactions involved in the deadlock. Thus for a cycle of length three, the number of messages required for the worst case would be two for the proposed algorithm and the Obermarck's algorithm would require tree messages. For a cycle of length four, the worst case would require five messages under the proposed algorithm vs. cycle of length five, nine messages would be required. We want to stress again that the worst case performance of the proposed algorithm only occurs, however, when all transactions involved migrate simultaneously, and the lexical ordering of the transactions is such that n-1 messages are sent on the first iteration. It is safe to assume that the worst case scenario does not always occur with each global deadlock and therefore the real performance of the proposed algorithm is expected to be better than we stated here. However, we must point out that the decrease in the number of inter-site

messages comes at the cost of slightly more complex lock tables and at the cost of each transaction carrying with it slightly more information (its Lock History). The amount of time used in level one activity is minimal, since only a single resource's Lock Table is used to determine the set of transactions whose Lock Histories must be combined. Even with level two, the time required to construct a WFG using all Wait For information at a site should take no longer than the construction of a WFG in Obermarck's algorithm. In [OBE82], Obermarck does not discuss the factors which trigger deadlock detection, but for this analysis, it is assumed that it is triggered X units of time after a transaction waits for a resource. His algorithm constructs a WFG at each iteration of the deadlock detection cycle, regardless of the potential size of the cycle. Since the proposed algorithm performs a comparable construction only when cycles of length two have essentially been eliminated as a possibility, it appears that the proposed algorithm will require less time to execute whenever it is invoked.

V. CONCLUSIONS

The proposed algorithm has been shown to be able to detect deadlock with smaller number of inter-site messages than any existing algorithm for deadlock detection in distributed computing systems. We have shown that for the deadlock scenarios analyzed in this paper the proposed algorithm requires from zero to N-1 (where $N = \sum_{k=1}^{n}(n-k)$) messages to detect a global deadlock, where n is the number of transactions and sites involved in the deadlock cycle. It requires no messages when the transaction migrations leading to the deadlock occur sequentially. This is because when a transaction migrates, it "brings along" a pertinent wait-for information from its current site. The worst case for the proposed algorithm is when the transactions involved migrate simultaneously. This can result in as many as N-1 messages, depending on the ordering of the unique transaction identifiers. Obermarck's algorithm for this case requires N

messages, which is always one message more than the number required by the proposed algorithm. The reason that the proposed algorithm requires one less iteration of message passing is because the Lock Histories of each transaction are brought along with the transaction when it migrates, and thus each site has more information than the sites would have using Obermarck's algorithm. The most important point is that the proposed algorithm can detect the most frequent deadlocks without any inter-site messages. The proposed algorithm requires no inter-site messages for direct deadlocks of cycle length two involving two sites, or for deadlocks of cycle length > 2 when a) a sequential migration of transactions in order of their lexically ordered unique identifiers has occured regardless of the number of transactions or sites involved or b) the deadlock is direct and involves only two sites where at one site only two transactions conflict and an arbitrary number of transactions conflict at the other site. For all other types of deadlocks the proposed algorithm requires one less message than Obermarck's algorithm. The proposed algorithm could be modified by combining levels one and two, if the number of resources and transactions in the system are small, and therefore the cost of creating WFG's at level 2 would be comparable to the cost of the level 1 WFG construction. The cost of construction of the WFG's used by the algorithm could be saved by not constructing them at all, but merely examining the WFS's and Lock Histories, since all required information is contained in them. The delays which have been built-in to the algorithm can be adjusted empirically to determine the optimum delays for a particular implementation.

REFERENCES

[GLI80] Gligor, V. and Shattuck, S., "On Deadlock Detection in Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. SE-6, p. 435-440, September 1980.

[GOL77] Goldman, B. "Deadlock Detection in Computer Networks", Massachusetts Institute of Technology Technical Report TR-MIT/LCS/TR-185, September, 1977.

[GRA78] Gray, J. "Notes on Data Base Operating Systems", IBM Research Division Research Report RJ2188(30001), February, 1978.

[GRA81A] Gray, J., Homan, P., Korth, H. and Obermarck, R. "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Distributed Database System", paper presented at *5th Berkeley Workshop*, on Distributed Data Management and Computer Networks, San Francisco, February 1981.

[GRA81B] Gray, J. "The Transaction Concept: Virtues and Limitations", Tandem Technical Report TR81.3, June 1981.

[MEN79] Menasce, D. and Muntz, R., "Locking and Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering*, Vol. SE-5, No 3, p. 195-202, May 1979.

[OBE82] Obermarck, R. "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp. 187-209, June 1982.

[TSA82] Tsai and Belford, G., "Detecting Deadlock in a Distributed System", *Proceedings INFOCOM*, Las Vegas, 1 April 1982.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93940	30
Dushan Z. Badal, Code 52Zd Department of Computer Science Naval Postgraduate School Monterey, CA 93940	5
Robert B. Grafton Office of Naval Research Code 437 300 N. Quincy Street Arlington, VA 22217	1
David W. Mizell Office of Naval Research 1030 East Green Street Pasadena, CA 91106	2
CDR R. Ohlander DARPA 1400 Wilson Blvd. Arlington, VA 22209	1
Col. D. Adams DARPA 1400 Wilson Blvd. Arlington, VA 22209	1
CAPT W. Price AFOSR/NM Bolling AFB, D.C. 20332	1

Col. R. Schell 1
National Security Agency
C1
Fort George Meade, MD 20755

LCDR Mike Gehl 1
Elex 814A Naval Electronic Systems Command
Washington, D.C. 20363

CDR Tom Pigoski 1
3801 Nebraska Ave., NW
Washington, D.C. 20390

Dr. John Schiell 1
Naval Ocean Systems Center
Code 8321
San Diego, CA 92152